AVF Control Number: AVF-VSR-222.0589
88-09-23-RRS

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890113W1.10025
R.R. Software, Inc.
IntegrAda 4.1.0
KayPro 386 Host and Target

Completion of On-Site Testing:
13 January 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB, OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

F Implementation Dependencies

This appendix specifies certain system-dependent characteristics of IntegrAda, version 4.1.

F.1 Implementation Dependent Pragmas

In addition to the required Ada pragmas, IntegrAda also provides several others. Some of these pragmas have a *textual range*. Such pragmas set some value of importance to the compiler, usually a flag that may be On or Off. The value to be used by the compiler at a given point in a program depends on the parameter of the most recent relevant pragma in the text of the program. For flags, if the parameter is the identifier On, then the flag is on; if the parameter is the identifier Off, then the flag is off; if no such pragma has occurred, then a default value is used.

The range of a pragma - even a pragma that usually has a textual range - may vary if the pragma is not inside a compilation unit. This matters only if you put multiple compilation units in a file. The following rules apply:

1) If a pragma is inside a compilation unit, it affects only that unit;
2) If a pragma is outside a compilation unit, it affects all following compilation units in the compilation.

Certain required Ada pragmas, such as INLINE, would follow different rules; however, as it turns out, IntegrAda ignores all pragmas that would follow different rules.

The following system-dependent pragmas are defined by IntegrAda. Unless otherwise stated, they may occur anywhere that a pragma may occur.

ALL_CHECKS Takes one of two identifiers On or Off as its argument, and has a textual range. If the argument is Off, then this pragma causes suppression of arithmetic checking (like pragma ARITHCHECK - see below), range checking (like pragma RANGECHECK - see below), storage error checking, and elaboration checking. If the argument is On, then these checks are all performed as usual. Note that pragma ALL_CHECKS does not affect the status of the DEBUG pragma; for the fastest run time code (and the worst run time checking), both ALL_CHECKS and DEBUG should be turned Off and the pragma OPTIMIZE (Time) should be used. Note also that ALL_CHECKS does not affect the status of the ENUMTAB pragma. Combining check suppression using the pragma ALL_CHECKS and using the pragma SUPPRESS may cause unexpected results; it should not be done.

However, ALL_CHECKS may be combined with the IntegrAda pragmas ARITHCHECK and RANGECHECK; whichever relevant pragma has occurred most recently will determine whether a given check is performed. ALL_CHECKS is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

ARITHCHECK Takes one of the two identifiers On or Off as its argument, and has a textual range. Where ARITHCHECK is on, the compiler is permitted to (and generally does) not generate checks for situations where it is permitted to raise NUMERIC_ERROR; these checks include overflow checking and checking for division by zero. Combining check suppression using the pragma ARITHCHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, ARITHCHECK may be combined with the IntegrAda pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. ARITHCHECK is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

CLEANUP Takes an integer literal in the range 0..3 as its argument, and has a textual range. Using this pragma allows the IntegrAda run-time system to be less than meticulous about recovering temporary memory space it uses. This pragma can allow for smaller and faster code, but can be dangerous; certain constructs can cause memory to be used up very quickly. The smaller the parameter, the more danger is permitted. A value of 3 - the default value - causes the run-time system to be its usual immaculate self. A value of 0 causes no reclamation of temporary space. Values of 1 and 2 allow compromising between "cleanliness" and speed. Using values other than 3 adds some risk of your program running out of memory, especially in loops which contain certain constructs.

DEBUG

Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of line number code and procedure name code. When DEBUG is on, such code is generated. When DEBUG is off, no line number code or procedure names are generated. This information is used by the walkback which is generated after a run-time error (e.g., an unhandled exception). The walkback is still generated when DEBUG is off, but the line numbers will be incorrect, and no subprogram names will be printed. DEBUG's initial state can be set by the command line; if no explicit option is given, then DEBUG is initially on. Turning DEBUG off saves space, but causes the loss of much of IntegrAda's power in describing run time errors.

Notes:
DEBUG should only be turned off when the program has no errors. The information provided on an error when DEBUG is off is not very useful.

If DEBUG is on at the beginning of a subprogram or package specification, then it must be on at the end of the specification. Conversely, if DEBUG is off at the beginning of such a specification, it must be off at the end. If you want DEBUG to be off for an entire compilation, then you can either put a DEBUG pragma in the context clause of the compilation or you can use the appropriate compiler option.

ENUMTAB

Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of enumeration tables. Enumeration tables are used for the attributes IMAGE, VALUE, and WIDTH, and hence to input and output enumeration values. The tables are generated when ENUMTAB is on. The state of the ENUMTAB flag is significant only at enumeration type definitions. If this pragma is used to prevent generation of a type's enumeration tables, then using the three mentioned attributes causes an erroneous program, with unpredictable results; furthermore, the type should not be used as a generic actual discrete type, and in particular TEXT_IO.ENUMERATION_IO should not be instantiated for the type. If the enumeration type is not needed for any of these purposes, the tables, which use a lot of space, are unnecessary. ENUMTAB is on by default.

PAGE_LENGTH

This pragma takes a single integer literal as its argument. It says that a page break should be added to the listing after each occurrence of the given number of lines. The default page length is 32000, so that no page breaks are generated for most programs. Each page starts with a header that looks like the following:

IntegrADA Version 4.1 compiling *file* on *date* at *time*

RANGECHECK Takes one of the two identifiers On or Off as its argument, and has a textual range. Where RANGECHECK is off, the compiler is permitted to (and generally does) not generate checks for situations where it is expected to raise CONSTRAINT_ERROR; these checks include null pointer checking, discriminant checking, index checking, array length checking, and range checking. Combining check suppression using the pragma RANGECHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, RANGECHECK may be combined with the IntegrAda pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. RANGECHECK is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

SYSLIB This pragma tells the compiler that the current unit is one of the standard IntegrAda system libraries. It takes as a parameter an integer literal in the range 1 .. 15; only the values 1 through 4 are currently used. For example, system library number 2 provides floating point support. Do not use this pragma unless you are writing a package to replace one of the standard IntegrAda system libraries.

VERBOSE Takes On or Off as its argument, and has a textual range. VERBOSE controls the amount of output on an error. If VERBOSE is on, the two lines preceding the error are printed, with an arrow pointing at the error. If VERBOSE is off, only the line number is printed.

VERBOSE(Off):

```
Line 16 at Position 5
*ERROR* Identifier is not defined
```

VERBOSE(On):

```
 15: if X = 10 then
 16:     Z := 10;
---------^
*ERROR* Identifier is not defined
```

The reason for this option is that an error message with VERBOSE on can take a long time to be generated, especially in a large program. VERBOSE's initial condition can be set by the compiler command line.

Several required Ada pragmas may have surprising effects in IntegrAda. The PRIORITY pragma may only take the value 0, since that is the only value in the range System.Priority. Specifying any OPTIMIZE pragma turns on optimization; otherwise, optimization is only done if specified on the compiler's command line. The SUPPRESS pragma is ignored unless it only has one parameter. Also, the following pragmas are always ignored: CONTROLLED, INLINE, MEMORY_SIZE, PACK, SHARED, STORAGE_UNIT, and SYSTEM_NAME. Pragma CONTROLLED is always ignored because IntegrAda does no automatic garbage collection; thus, the effect of pragma CONTROLLED already applies to all access types. Pragma SHARED is similarly ignored: IntegrAda's non-preemptive task scheduling gives the appropriate effect to all variables. The pragmas INLINE, PACK, and SUPPRESS (with two parameters) all provide recommendations to the compiler; as Ada allows, the recommendations are ignored. The pragmas MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME all attempt to make changes to constants in the System package; in each case, IntegrAda allows only one value, so that the pragma is ignored.

F.2 Implementation Dependent Attributes

IntegrAda does not provide any attributes other than the required Ada attributes.

F.3 Specification of the Package SYSTEM

The package System for IntegrAda has the following definition.

```
package System is

    -- System package for IntegrAda

    -- Types to define type Address.
    type Word is range 0 .. 65536;
    for Word'Size use 16;
    type Offset_Type is new Word;
    type Address is record
        Offset : Offset_Type;
        Segment : Word;
    end record;
    Function "+" (Left : Address; Right : Offset_Type) Return Address;
    Function "+" (Left : Offset_Type; Right : Address) Return Address;
    Function "-" (Left : Address; Right : Offset_Type) Return Address;
    Function "-" (Left, Right : Address) Return Offset_Type;

    type Name is (MS_DOS2);

    System_Name : constant Name := MS_DOS2;

    Storage_Unit : constant := 8;
    Memory_Size : constant := 65536;
        -- Note: The actual memory size of a program is determined
        -- dynamically; this is the maximum number of bytes in the data
        -- segment.

    -- System Dependent Named Numbers:
    Min_Int : constant := -2_147_483_648;
    Max_Int : constant :=  2_147_483_647;
    Max_Digits : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta : constant := 2#1.0#E-31;
        -- equivalently, 4.656612873077392578125E-10
    Tick : constant := 0.01; -- Some machines have less accuracy;
        -- for example, the IBM PC actually ticks about
        -- every 0.06 seconds.

    -- Other System Dependent Declarations
    subtype Priority is Integer range 0..0;

    type Byte is range 0 .. 255;
    for Byte'Size use 8;
```

```
end System;
```

The type Byte in the System package corresponds to the 8-bit machine byte. The type Word is a 16-bit Unsigned Integer type, corresponding to a machine word.

F.4 Restrictions on Representation Clauses

If T is a discrete type, or a fixed point type, then the size expression can give any value between 1 and 1000 bits (subject, of course, to allowing enough bits for every possible value). For other types, the expression must give the default size for T.

A length clause that specifies T'STORAGE_SIZE for an access type is not supported; IntegrAda uses a single large common heap.

A length clause that specifies T'STORAGE_SIZE for a task type T is supported. Any integer value can be specified. Values smaller than 256 will be rounded up to 256 (the minimum T'Storage_Size), as the Ada standard does not allow raising an exception in this case.

A length clause that specifies T'SMALL for a fixed point type must give a value (subject to the Ada restrictions) in the range

```
2.0 ** (-99) .. 2.0 ** 99,
```

inclusive.

An enumeration representation clause for a type T may give any integer values within the range System.Min_Int .. System.Max_Int. If a size length clause is not given for the type, the type's size is determined from the literals given. (If all of the literals fit in a byte, then Byte'Size is used; similarly for Integer and Long_Integer).

The expression in an alignment clause in a record representation clause must equal 1.

A component clause must give a storage place that is equivalent to the default value of the POSITION attribute for such a component.

A component clause must give a range that starts at zero and extends to one less than the size of the component.

IntegrAda supports address clauses on most objects. Address clauses are not allowed on parameters, generic formal parameters, and renamed objects. The address given for an object address clause may be any legal value of type System.Address. It will be interpreted as an absolute machine address, using the segment part as a selector if in the protected mode. It is the user's responsibility to ensure that the value given makes sense (i.e., points at memory, does not overlay other objects, etc.) No other address clauses are supported.

F.5 Implementation Defined Names

IntegrAda uses no implementation generated names.

F.6 Address Clause Expressions

The address given for an object address clause may be any legal value of type System.Address. It will be interpreted as an absolute machine address, using the segment part as a selector if in the protected mode. It is the user's responsibility to ensure that the value given makes sense (i.e., points at memory, does not overlay other objects, etc.)

F.7 Unchecked_Conversion Restrictions

We first make the following definitions:
A type or subtype is said to be a *simple type* or a *simple subtype* (respectively) if it is a scalar (sub)type, an access (sub)type, a task (sub)type, or if it satisfies the following two conditions:

1) If it is an array type or subtype, then it is constrained and its index constraint is static; and
2) If it is a composite type or subtype, then all of its subcomponents have a simple subtype.

A (sub)type which does not meet these conditions is called *non-simple*. Discriminated records can be simple; variant records can be simple. However, constraints which depend on discriminants are non-simple (because they are non-static).

IntegrAda imposes the following restriction on instantiations of Unchecked_Conversion: for such an instantiation to be legal, both the source

actual subtype and the target actual subtype must be simple subtypes, and they must have the same size.

F.8 Implementation Dependencies of I/O

The syntax of an external file name depends on the operating system being used. Some external files do not really specify disk files; these are called *devices*. Devices are specified by special file names, and are treated specially by some of the I/O routines.

The syntax of an MS-DOS 2.xx or 3.xx filename is:

```
[d:][path]filename[.ext]
```

where "d:" is an optional disk name; "path" is an optional path consisting of directory names, each followed by a backslash; "filename" is the filename (maximum 8 characters); and ".ext" is the extension (or file type). See your MS-DOS manual for a complete description. In addition, the following special device names are recognized:

STI: MS-DOS standard input. The same as Standard_Input. Input is buffered by lines, and all MS-DOS line editing characters may be used. Can only be read.

STO: MS-DOS standard output. The same as Standard_Output. Can only be written.

ERR: MS-DOS standard error. The output to this device cannot be redirected. Can only be written.

CON: The console device. Single character input with echoing. Due to the design of MS-DOS, this device *can* be redirected. Can be read and written.

AUX: The auxiliary device. Can be read or written.

LST: The list (printer) device. Can only be written.

KBD: The console input device. No character interpretation is performed, and there is no character echo. Again, the input to this device can be redirected, so it does not *always* refer to the physical keyboard.

The MS-DOS device files may also be used (CON, AUX, and PRN without colons ':'). For compatibility reasons, we do not recommend the use of these names.

The MS-DOS 2.xx version of the I/O system will do a search of the default search path (set by the DOS PATH command) if the following conditions are met:

1) No disk name or path is present in the file name; and
2) The name is not that of a device.

Alternatively, you may think of the search being done if the file name does not contain any of the characters ':', '/', or '\'.

The default search path cannot be changed while the program is running, as the path is copied by the IntegrAda program when it starts running.

Note:
Creates will never cause a path search as they must work in the current directory.

Upon normal completion of a program, any open external files are closed. Nevertheless, to provide portability, we recommend explicitly closing any files that are used.

Sharing external files between multiple file objects causes the corresponding external file to be opened multiple times by the operating system. The effects of this are defined by your operating system. This external file sharing is only allowed if all internal files associated with a single external file are opened only for reading (mode In_File), and no internal file is Created. Use_Error is raised if these requirements are violated. A Reset to a writing mode of a file already opened for reading also raise Use_Error if the external file also is shared by another internal file.

Binary I/O of values of access types will give meaningless results and should not be done. Binary I/O of types which are not simple types (see definition in Section F.7, above) will raise Use_Error when the file is opened. Such types require specification of the block size in the form, a capability which is not yet supported.

The form parameter for Sequential_IO and Direct_IO is always expected to be the null string.

The type Count in the generic package Direct_IO is defined to have the range 0 .. 32767.

Ada specifies the existence of special markers called *terminators* in a text file. IntegrAda defines the line terminator to be <LF> (line feed), with or without an

additional <CR> (carriage return). The page terminator is the <FF> (form feed) character; if it is not preceded by a <LF>, a line terminator is also assumed.

The file terminator is the end-of-file returned by the host operating system. If no line and/or page terminator directly precedes the file terminator, they are assumed. If the form "Z" is used, the <Ctrl>-Z character also represents the end-of-file. This form is not necessary to correctly read files produced with IntegrAda and most other programs, but may be occasionally necessary. The only legal forms for text files are "" (the null string) and "Z". All other forms raise USE_ERROR.

If the form is "", the <Ctrl>-Z character is ignored on input. The <CR> character is always ignored on input. (They will *not* be returned by Get, for instance). All other control characters are sent directly to the user. Output of control characters does not affect the layout that Text_IO generates. In particular, output of a <LF> before a New_Page does not suppress the New_Line caused by the New_Page.

On output, the "Z" form causes the end-of-file to be marked by a <Ctrl>-Z; otherwise, no explicit end-of-file character is used. The character pair <CR> <LF> is written to represent the line terminator. Because <CR> is ignored on input, this is compatible with input.

The type Text_IO.Count has the range 0 .. 32767; the type Text_IO.Field also has the range 0 .. 32767.

IO_Exceptions.USE_ERROR is raised if something cannot be done because of the external file system; such situations arise when one attempts:

- to create or open an external file for writing when the external file is already open (via a different internal file).
- to create or open an external file when the external file is already open for writing (via a different internal file).
- to reset a file to a writing mode when the external file is already open (via a different internal file).
- to write to a full disk (Write, Close);
- to create a file in a full directory (Create);
- to have more files open than the OS allows (Open, Create);
- to open a device with an illegal mode;
- to create, reset, or delete a device;
- to create a file where a protected file (i.e., a directory or read-only file) already exists;
- to delete a protected file;
- to use an illegal form (Open, Create), or
- to open a file for a non-simple type without specifying the block size;
- to open a device for direct I/O.

IO_Exceptions.DEVICE_ERROR is raised if a hardware error other than those covered by USE_ERROR occurs. These situations should never occur, but may on rare occasions. For example, DEVICE_ERROR is raised when:

- a file is not found in a Close or a Delete;
- a seek error occurs on a direct Read or Write; or
- a seek error occurs on a sequential End_Of_File.

The subtypes Standard.Positive and Standard.Natural, used by some I/O routines, have the maximum value 32767.

No package Low_Level_IO is provided.

F.9 Running the compiler and linker

The IntegrAda compiler is invoked using the following format:

```
COMPILE [d:] filename [.ext] {/option}
```

where filename is an MS/DOS file name with optional disk name [d:], optional extension [.ext], and compiler options {/option}. If no disk name is specified, the current disk is assumed. If no extension is specified, .PKG is assumed.

The compiler options are:

B Brief error messages. The line in error is not printed (equivalent to turning off pragma VERBOSE).

D Don't generate debugging code (equivalent to turning off pragma DEBUG)

F Use in-line 8087 instructions for Floating point operations. By default the compiler generates library calls for floating point operations. The 8087 may be used to execute the library calls. A floating point support library is still required, even though this option is used.

L Create a listing file with name filename.PRN on the same disk as filename. The listing file will be a listing of only the last compilation unit in a file.

Ld Create a listing file on specified disk 'd'. Choices are 'A' through 'W'.

Ox Object code memory model. X is 0 or 1. Memory model 0 creates faster, smaller code, but limits all code in all units of a program to one MS-DOS segment (i.e., 64 kilobytes); Memory model 1 allows code size limited only by your machine and operating system. See the linker (BIND) manual for more information. Memory model 0 is assumed if this option is not given. The compiler records the memory model for which each library unit was compiled, and it will complain if any mismatches occur. Thus, the compiler enforces that if it is run using the /o1 option, then all of the withed units must have been compiled with the same option.

Q Quiet error messages. This option causes the compiler not to wait for the user to interact after an error. In the usual mode, the compiler will prompt the user after each error to ask if the compilation should be aborted. This option is useful if the user wants to take a coffee break while the compiler is working, since all user prompts are suppressed. The errors (if any) will not stay on the screen when this option is used; therefore, the console traffic should be sent to the printer or to a file. Be warned that certain syntax errors can cause the compiler to print many error messages for each and every line in the program. A lot of paper could be used this way! Note that the /Q option disallows disk swapping, even if the /S option is given.

Rd Route the JRL file to the specified disk 'd'. Choices are 'A' through 'W'. The default is the same disk as filename.

Sd Route Scratch files to specified disk. This option is useful if you have a RAM disk or if your disk does not have much free space. The use of this option also allows disk swapping to load package specification (.SYM) files. Normally, after both the compiler and source file disks are searched for .SYM files, an error is produced if they are not all found. However, when the /S option is used, the compiler disk may be removed and replaced by a disk to search. The linker has a similar option, which allows the development of large programs on systems with a small disk capacity. Note that disk swapping is *not* enabled by the /S option if the /Q (quiet option) is also given. The /Q option is intended for batch mode compiles, and its purpose conflicts with the disk swapping. The main problem is that when the /S option is used to put scratch files on a RAM disk, a batch file may stop waiting for a missing .SYM or ERROR.MSG file; such behavior would not be appropriate when /Q is specified.

T Generate information which allows trimming unused subprograms from the code. This option tells the compiler to generate information which can be used by the remove subprograms from the final code. This option increases the size of the .JRL files produced. We recommend that it be used on reusable libraries of code (like trig. libraries or stack packages) - that is those compilations for which it is likely that some subprograms are not called.

W Don't print any warning messages. For more control of warning messages, use the following option form (Wx).

Wx Print only warnings of level less than the specified digit 'x'. The given value of x may be from 1 to 9. The more warnings you are willing to see, the higher the number you should give.

X Handle eXtra symbol table information. This is for the use of debuggers and other future tools. This option requires large quantities of memory and disk space, and thus should be avoided if possible.

Z Turn on optimization. This has the same effect as if the pragma OPTIMIZE were set to SPACE throughout your compilation.

The default values for the command line options are:

B Error messages are verbose.
D Debug code is generated.
F Library calls are generated for floating point operations.
L No listing file is generated.
O Memory model 0 is used.
Q The compiler prompts for abort after every error.
R The JRL file is put on the same disk as the input file.
S Scratch files are put on the same disk as the compiler.
T No trimming code is produced.
W All warnings are printed.
X Extra symbol table information is not generated.
Z Optimization is done only where so specified by pragmas.

Leading spaces are disregarded between the filename and the call to COMPILE. Spaces are otherwise not recommended on the command line. The presence of blanks to separate the options or between the filename and the extension will be ignored.

Examples:

```
COMPILE test/Q/L
COMPILE test.run/W4
COMPILE test
COMPILE test   .run /B /W/L
```

The compiler produces a SYM (SYMbol table information) file when a specification is compiled, and a SRL or JRL (Specification ReLocatable or Janus ReLocatable) file when a body is compiled. To make an executable program, the appropriate SRL and JRL files must be *linked* (combined) with the run-time libraries. This is accomplished by running the IntegrAda linker, BIND.

The IntegrAda linker is invoked using the following format:

```
BIND [d:] filename [/option]
```

Here "filename" is the name of the SRL or JRL file created when the main program was compiled (without the .SRL or .JRL extension) with optional disk name [d:], and compiler options [/option]. The filename usually corresponds to the first eight letters of the name of your main program. A disk may be specified where the files are to be found. See the linker manual for more detailed directions. We summarize here, however, a few of the most commonly used linking options:

E Create an EXE file. This is assumed if the /O1 option is given. This allows allow a slightly larger total program size if memory model is used.
F0 Use software floating point (the default).
F2 Use hardware (8087) floating point.
L Display lots of information about the loading process.
O0 Use memory model 0 (the default); see the description of the /O option in the compiler, above.
O1 Use memory model 1.
Q Use quiet error messages; i.e., don't wait for the user to interact after an error.
T Trim unused subprograms from the code. This option tells the linker to remove subprograms which are never called from the final output file. This option reduces space usage of the final file by as much as 30K.

Examples:

```
BIND test
BIND test /Q/L
BIND test/O1/L/F2
```

Note that if you do not have a hardware floating point chip, and if you are using memory model 0, then you generally will not need to use any linker options.

Appendix F: Implementation Dependencies

This page intentionally left blank